Trained ANN model

Compile the model

```
# # Compile the model
ann_model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

Compiling a model in Keras is an essential step in defining how the model will learn during training. The **compile()** method specifies the following components:

- **Loss Function**: Measures how well the model is performing. It quantifies the difference between the model's predictions and the actual target values. The goal of training is to minimize this loss. **binary_crossentropy** is chosen because in this project we want to identify whether the customers “churn” or “no churn” (binary classification problem).
    - At every iteration, the model calculates the loss for the current predictions.
    - The optimizer then adjusts the model's weights to reduce the loss.
- **Optimizer**: Determines how the model updates its weights during training to minimize the loss. adam is chosen because:
    - It is the default optimizer for most tasks
    - It combines momentum-based optimization and adaptive learning rates.
    - Requires minimal parameter tuning.
    - Suitable for most problems, including binary classification.
- **Metrics**: are used to evaluate the model's performance during training and testing. They don’t affect the training process itself but provide feedback about the model.
    - **accuracy** is a simple and intuitive metric for binary classification tasks like churn prediction. It measures the proportion of correct predictions.

Train the model

```
# # Train the model
history = ann_model.fit(
    X_train_scaled, y_train,
    validation_split=0.2,
    epochs=50,
    batch_size=32,
    callbacks=[tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)],
    verbose=1
)
```

Training a model in Keras involves using the **fit()** method, which handles the entire training process, including feeding the data into the model, computing the loss, updating weights using the optimizer, and tracking metrics.

- validation_split = 0.2 means 20% of X_train_scaled and y_train are used for validation
    - Splits a fraction of the training data into a validation set.
    - Validation data is used to monitor the model's performance on unseen data during training.
    - In other words: uses 80% of the training set (with an additional 20% reserved for validation).
- epochs = 50: Maximum of 50 passes over the training data.
    - Number of complete passes over the training data.
    - More epochs allow the model to learn longer, but too many epochs can lead to overfitting.
- batch_size = 32: Processes 32 samples at a time during training.
    - Number of samples processed before updating the model's weights.
    - Common values: 32, 64, or powers of 2
    - Smaller batches:
        - Converge more slowly but can generalize better.
        - Require less memory.
    - Larger batches:
        - Faster training but may lead to overfitting.
- callbacks: A list of functions to execute during training to enhance control.
    - EarlyStopping: Stops training when performance stops improving.
- verbose = 1: Controls the level of training information displayed:
    - 0: No output.
    - 1: Displays a progress bar for each epoch.
    - 2: Displays one line per epoch.

The fit() method returns a History object, which contains:

- Training metrics (e.g., loss, accuracy).
- Validation metrics (if validation data is provided).
- Metrics for each epoch.

We can print the History object for analysis as below:

```
print(history.history)
```

```
Epoch 1/50
139/139 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.6344 - loss: 0.6209 - val_accuracy: 0.7687 - val_loss: 0.4547
Epoch 2/50
139/139 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.7752 - loss: 0.4599 - val_accuracy: 0.7723 - val_loss: 0.4447
Epoch 3/50
139/139 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.7863 - loss: 0.4500 - val_accuracy: 0.7786 - val_loss: 0.4402
Epoch 4/50
139/139 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.7801 - loss: 0.4479 - val_accuracy: 0.7732 - val_loss: 0.4384
Epoch 5/50
139/139 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.7881 - loss: 0.4346 - val_accuracy: 0.7867 - val_loss: 0.4377
Epoch 6/50
139/139 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8040 - loss: 0.4150 - val_accuracy: 0.7831 - val_loss: 0.4378
Epoch 7/50
139/139 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8031 - loss: 0.4235 - val_accuracy: 0.7876 - val_loss: 0.4378
Epoch 8/50
139/139 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.7985 - loss: 0.4241 - val_accuracy: 0.7831 - val_loss: 0.4371
Epoch 9/50
139/139 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.7942 - loss: 0.4270 - val_accuracy: 0.7840 - val_loss: 0.4385
Epoch 10/50
139/139 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.7942 - loss: 0.4318 - val_accuracy: 0.7813 - val_loss: 0.4406
Epoch 11/50
139/139 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8011 - loss: 0.4201 - val_accuracy: 0.7849 - val_loss: 0.4356
Epoch 12/50
139/139 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.7963 - loss: 0.4302 - val_accuracy: 0.7885 - val_loss: 0.4365
Epoch 13/50
139/139 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.7917 - loss: 0.4335 - val_accuracy: 0.7822 - val_loss: 0.4355
Epoch 14/50
139/139 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8055 - loss: 0.4140 - val_accuracy: 0.7876 - val_loss: 0.4341
Epoch 15/50
139/139 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8002 - loss: 0.4242 - val_accuracy: 0.7858 - val_loss: 0.4340
Epoch 16/50
139/139 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.7974 - loss: 0.4079 - val_accuracy: 0.7885 - val_loss: 0.4330
Epoch 17/50
139/139 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.7993 - loss: 0.4201 - val_accuracy: 0.7858 - val_loss: 0.4337
Epoch 18/50
139/139 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.7986 - loss: 0.4283 - val_accuracy: 0.7867 - val_loss: 0.4332
Epoch 19/50
139/139 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.7926 - loss: 0.4338 - val_accuracy: 0.7849 - val_loss: 0.4322
Epoch 20/50
139/139 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8037 - loss: 0.4170 - val_accuracy: 0.7822 - val_loss: 0.4321
Epoch 21/50
139/139 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8064 - loss: 0.4129 - val_accuracy: 0.7867 - val_loss: 0.4308
```

Result of printing History

Evaluate on test data

```python
# # Evaluate the model on the test set
test_loss, test_accuracy = ann_model.evaluate(X_test_scaled, y_test, verbose=0)
print(f"Test Loss: {test_loss:.4f}")
print(f"Test Accuracy: {test_accuracy:.4f}")
```

Our model has been trained with X_train_scaled and y_train datasets. Now, in this step, we want to test the trained model using the X_test_scaled and y_test datasets.

Evaluating test data in Keras involves using the **evaluate()** method, which calculates the model's performance metrics (e.g., loss, accuracy) on a separate test dataset. This step ensures the model generalizes well to unseen data and gives a final measure of its performance.

The **evaluate()** method returns the loss and any metrics specified during **compile()**.

- **print(f"Test Loss: {loss:.4f}")**:
    - Prints the test loss rounded to 4 decimal places.

    - loss indicates how well the model's predictions match the actual labels on the test set.
    - Lower values are better, indicating the model is making accurate predictions.
- **print(f"Test Accuracy: {accuracy:.4f}")**:
    - Prints the test accuracy rounded to 4 decimal places.
    - Accuracy measures the fraction of correct predictions:
    - Higher values (closer to 1.0 or 100%) indicate better performance.

Result of the above print functions:

```
Test Loss: 0.4338
Test Accuracy: 0.7918
```

The results (loss: 0.4338, accuracy: 0.7918) are reasonable but it can be improved.